Group members: Eduard Koshkelyan, Marwane Bidamou, Roshan Maharjan

Problem 1

1. Explanation of the Problem with Examples

You are required to implement a stack (Last In First Out - LIFO) using a queue (First In First Out - FIFO).

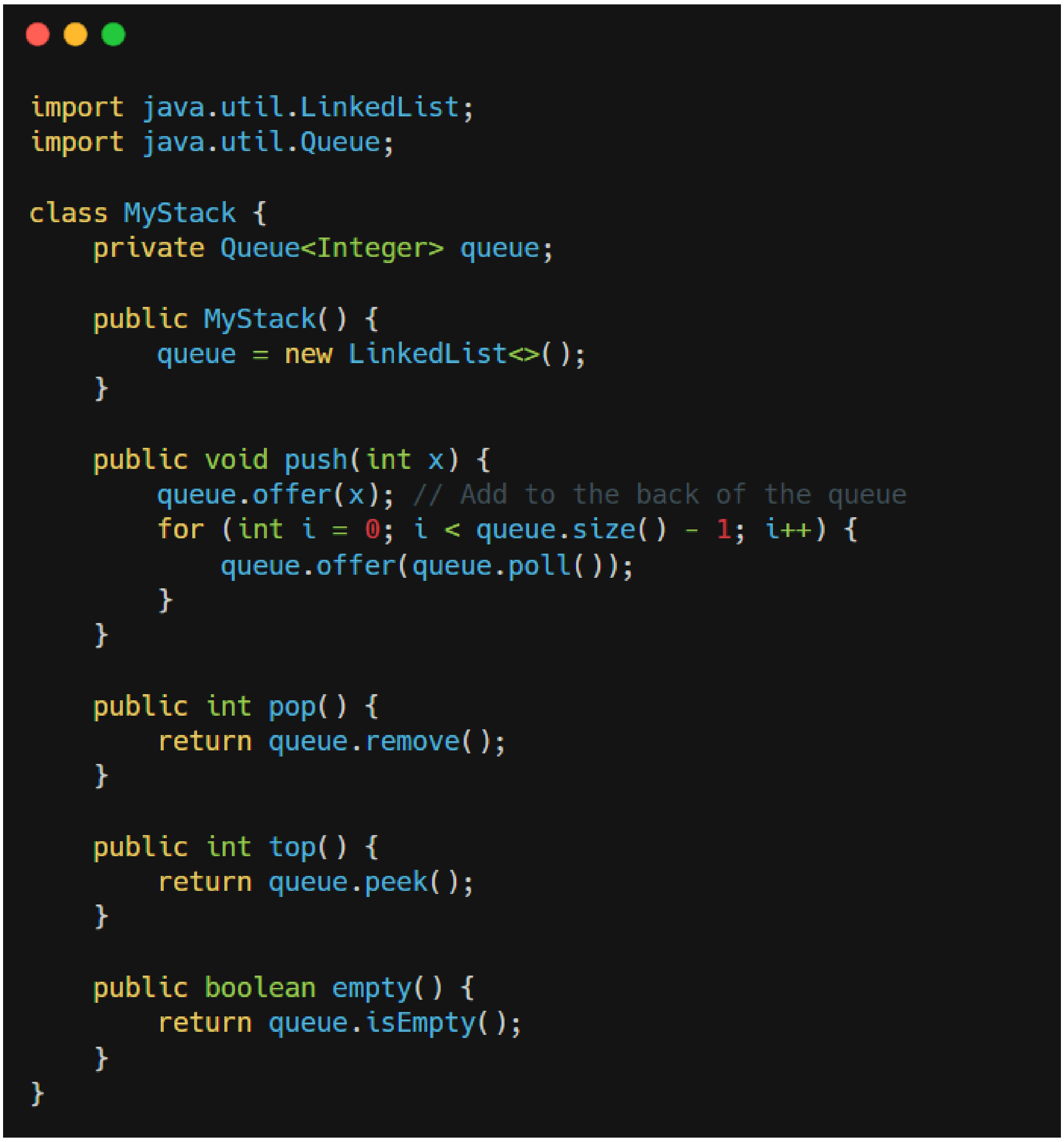
A stack supports the following operations:
1. push(x) – Push element x onto the stack.
2. pop() – Removes and returns the element on top of the stack.
3. top() – Returns the element on the top of the stack.
4. empty() – Returns whether the stack is empty.

2. Explanation of Possible Solutions

This implementation of a stack uses a **single queue** to maintain the LIFO (Last In First Out) order of a stack. After adding a new element to the queue using offer(x), the elements of the queue are rotated until the new element comes to the front, ensuring that the last element pushed becomes the first element retrieved.

- **Push Operation**:
    1. Add the new element to the queue.
    2. Rotate the queue by removing elements from the front and adding them to the back of the queue until the new element is at the front.
- **Pop Operation**:
    1. Remove and return the front element of the queue (which is the last added element due to the rotation in push).
- **Top Operation**:
    1. Return the front element of the queue (without removing it).
- **Empty Operation**:
    1. Check whether the queue is empty.

3. Implementation and Testing



```java
import java.util.LinkedList;
import java.util.Queue;

class MyStack {
    private Queue<Integer> queue;

    public MyStack() {
        queue = new LinkedList<>();
    }

    public void push(int x) {
        queue.offer(x); // Add to the back of the queue
        for (int i = 0; i < queue.size() - 1; i++) {
            queue.offer(queue.poll());
        }
    }

    public int pop() {
        return queue.remove();
    }

    public int top() {
        return queue.peek();
    }

    public boolean empty() {
        return queue.isEmpty();
    }
}
```

4. Time and Space Complexity of Your Algorithm

Push Operation:

- **Time Complexity**: O(n) – In each push operation, the new element is added to the queue in O(1), but then we need to rotate the entire queue (which involves n-1 operations), where n is the number of elements in the queue.
- **Space Complexity**: O(n) – The space used by the queue grows linearly with the number of elements.

Pop Operation:

- **Time Complexity**: O(1) – The pop operation just removes the front element of the queue, which takes constant time.
- **Space Complexity**: O(1) – No additional space is used in this operation.

Top Operation:

- **Time Complexity**: O(1) – The top operation only retrieves the front element, which is done in constant time.
- **Space Complexity**: O(1) – No additional space is used in this operation.

Empty Operation:

- **Time Complexity**: O(1) – The empty check is done in constant time.
- **Space Complexity**: O(1) – No additional space is used in this operation.

Problem 2

1. Explanation of the Problem with Examples

You are given the head of a singly linked list, and you need to reverse the list and return the reversed list's head.

- Example 1:
  - Input: 1 -> 2 -> 3 -> 4 -> 5 -> NULL
  - Output: 5 -> 4 -> 3 -> 2 -> 1 -> NULL

- Example 2:
  - Input: NULL
  - Output: NULL

- Example 3:
  - Input: 1 -> NULL
  - Output: 1 -> NULL

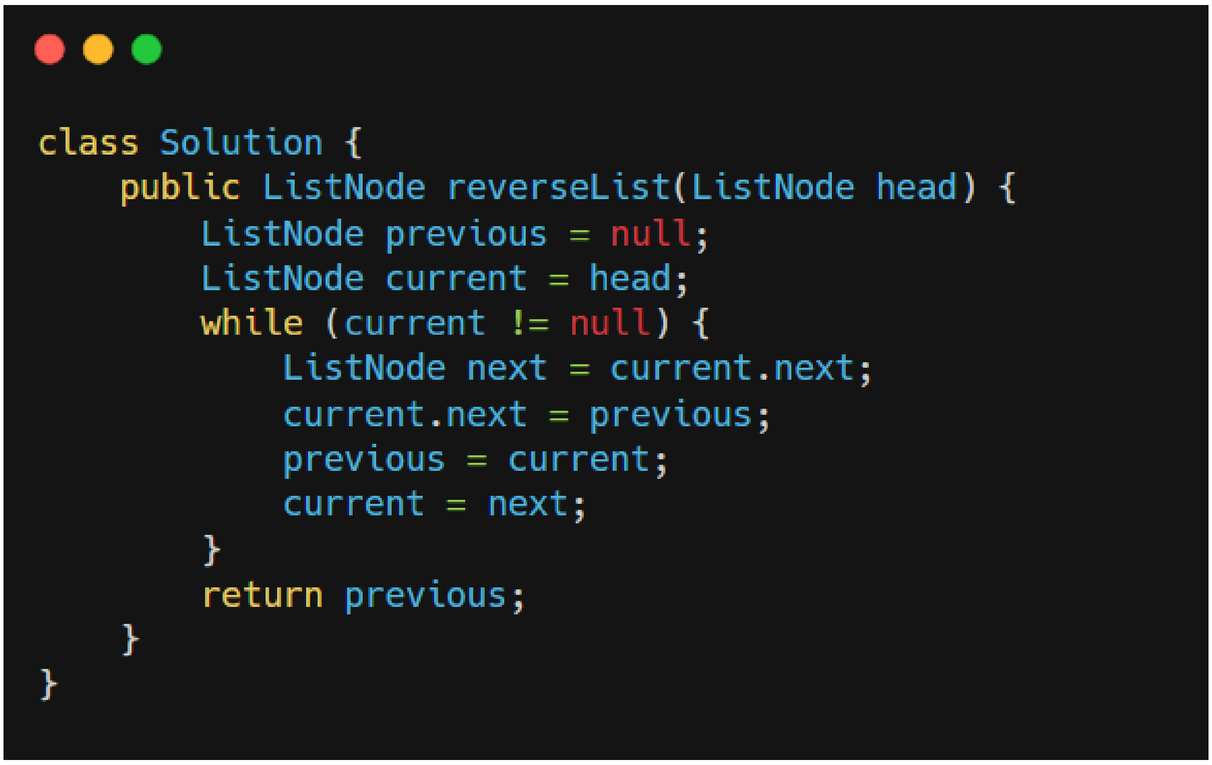
2. Explanation of Possible Solutions

Iterative Solution:

1. Initialize three pointers: previous (which starts as NULL), current (which starts as the head), and next.
2. Traverse the list and at each step, reverse the link by pointing current.next to previous.
3. Move previous and current one step forward.
4. Once current becomes NULL, previous will be the new head of the reversed list.

Group members: Eduard Koshkelyan, Marwane Bidamou, Roshan Maharjan

3. Implementation and Testing



```java
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode previous = null;
        ListNode current = head;
        while (current != null) {
            ListNode next = current.next;
            current.next = previous;
            previous = current;
            current = next;
        }
        return previous;
    }
}
```

4. Time and Space Complexity of Your Algorithm

Time Complexity: O(n) – Traversing the linked list exactly once.
Space Complexity: O(1) – Only use a few pointers, and no additional space is required.

Group members: Eduard Koshkelyan, Marwane Bidamou, Roshan Maharjan

Problem 3

1. Explanation of the Problem with Examples

You are given two strings, s and t, and you need to determine if t is an anagram of s. An anagram is a word formed by rearranging the letters of another, using all the original letters exactly once.

- Example 1:
  - Input: s = "anagram", t = "nagaram"
  - Output: true

- Example 2:
  - Input: s = "rat", t = "car"
  - Output: false

- Example 3:
  - Input: s = "listen", t = "silent"
  - Output: true

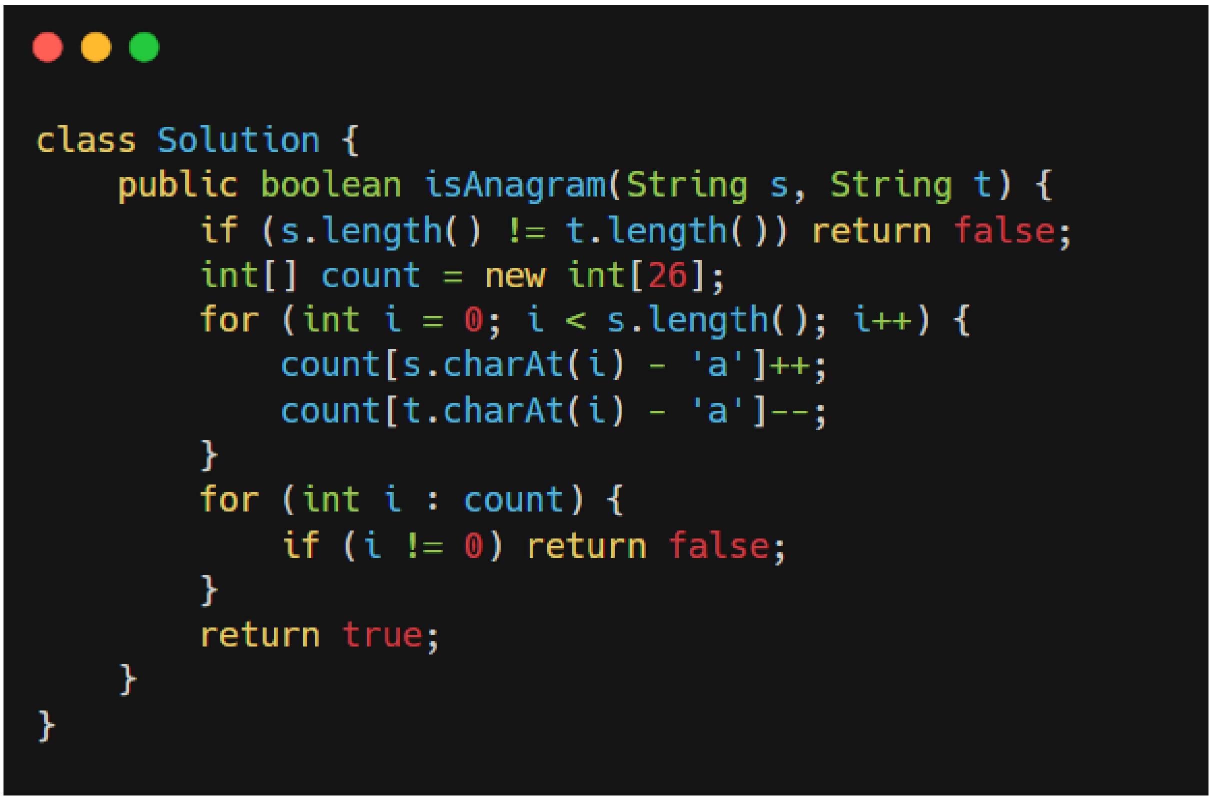
2. Explanation of Possible Solutions

Frequency Counting:

1. Create an array of size 26 (for each letter in the alphabet).
2. Increment the count for each character in the first string.
3. Decrement the count for each character in the second string.
4. If all counts are zero at the end, the strings are anagrams.

Group members: Eduard Koshkelyan, Marwane Bidamou, Roshan Maharjan

3. Implementation and Testing



```java
class Solution {
    public boolean isAnagram(String s, String t) {
        if (s.length() != t.length()) return false;
        int[] count = new int[26];
        for (int i = 0; i < s.length(); i++) {
            count[s.charAt(i) - 'a']++;
            count[t.charAt(i) - 'a']--;
        }
        for (int i : count) {
            if (i != 0) return false;
        }
        return true;
    }
}
```

4. Time and Space Complexity of Your Algorithm

- Frequency Count Solution:
  - Time Complexity: O(n) – Traversing the strings exactly once.
  - Space Complexity: O(1) – The space needed for the fixed-size frequency array is constant.